A Parallel Complexity Model for Functional Languages

Guy Blelloch John Greiner

October 20, 1994
CMU-CS-94-196

Carnegie Mellon

A Parallel Complexity Model for Functional Languages

Guy Blelloch John Greiner

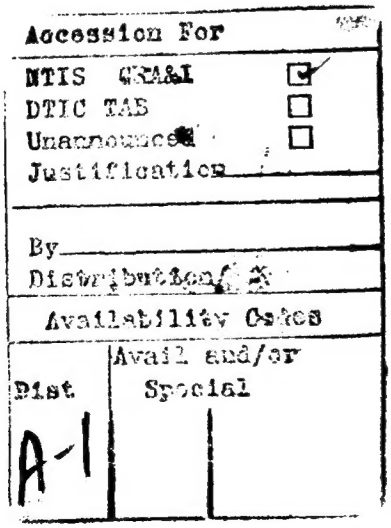
October 20, 1994
CMU-CS-94-196

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

A complexity model based on the λ-calculus with an appropriate operational semantics in presented and related to various parallel machine models, including the PRAM and hypercube models. The model is used to study parallel algorithms in the context of "sequential" functional languages, and to relate these results to algorithms designed directly for parallel machine models. For example, the paper shows that equally good upper bounds can be achieved for merging two sorted sequences in the pure λ-calculus with some arithmetic constants as in the EREW PRAM, when they are both mapped onto a more realistic machine such as a hypercube or butterfly network. In particular for n keys and p processors, they both result in an $O(n/p + \log^2 p)$ time algorithm. These results argue that it is possible to get good parallelism in functional languages without adding explicitly parallel constructs. In fact, the lack of random access seems to be a bigger problem than the lack of parallelism.

This research was sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330 and contract number F19628-91-C-0168. It was also supported in part by an NSF Young Investigator Award and by Finmeccanica.
The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Wright Laboratory or the U. S. Government.



Keywords: Functional languages, computer architecture, parallel algorithms, lambda calculus, models of computation

1 Introduction

Over the years many researchers have argued that an important aspect of functional languages is their inherent parallelism—since the languages lack side effects, it is safe to evaluate subexpressions in parallel. Furthermore researchers have presented many implementation techniques to take advantage of this parallelism, including data-flow [24], parallel graph reduction [17, 26], and various compiler techniques [11]. Such work has suggested that it might not be necessary to add explicit parallel constructs to functional languages to get adequate parallelism from functional languages.

There has been little study, however, of how much parallelism can be achieved for various problems, or how the inherent parallelism in functional languages relates to more standard models used for analyzing parallel algorithms, such as the PRAM. For example, what are asymptotic bounds for sorting using a parallel implementation of a functional language such as ML or Haskell? What kind of sort would we use? How would the bounds compare with parallel sorting algorithms designed for various machine models? Does it matter whether the language is strict or lazy? Before these can be answered, we first need to augment functional languages with a formal model of complexity. Furthermore, if we want to compare results to previous research on parallel algorithms, we also need to relate this complexity to run time on various machine models. This relation needs to capture some aspects of the parallel implementation of the language. To address these issues this paper makes the following contributions:

1. We introduce a parallel model based on the pure λ-calculus with applicative order evaluation and specified in terms of a profiling semantics [33, 34]. Complexity is given in terms of the total *work* executed by a program along with the depth (*steps*) of the computation, assuming that the two expressions of an application $e_1\ e_2$ are evaluated in parallel. We show that the model is basically equivalent within constant factors to the functional subsets of eager languages such as ML or Lisp when the parallelism in those languages comes from evaluating arguments in parallel. This correspondence allows us to prove our results for mapping the model onto various machines models using the simpler λ-calculus while allowing us to prove results on algorithms using an ML-like language.

2. We prove results on how the complexities in our model relate to complexities of various machine-based models, including the PRAM [12], hypercube, and butterfly models. The results are summarized in Figure 1. The proofs involve introducing a parallel version of the SECD machine [21], the P-ECD machine. A state of the P-ECD machine consists of a set of ECD substates, and each state transition of the machine transforms this set into a new set of substates. On each step the substates are scheduled across the processors of the host machine. We also prove results for simulating the PRAM model on our model.

3. We prove upper bounds in the model for merging and sorting. In particular we give a parallel algorithm that merges two sorted sequences of size n stored as balanced trees with $O(n)$ work and $O(\log n)$ steps. The algorithm borrows ideas from algorithms designed for the PRAM [35], but has some substantial changes to make up for the lack of random access. Based on this algorithm we can sort a sequence stored as a balanced tree with $O(n \log n)$ work and $O(\log^2 n)$ steps. For sequences stored as a list any algorithm would require $\Omega(n)$ steps just to traverse the list. This accentuates the importance of storing data as trees rather than lists to take advantage of parallel implementations of functional languages. Our work bounds are optimal for both merging and sorting and our step bounds are optimal for merging. Furthermore when the complexity for merging is mapped onto a hypercube or butterfly network, the resulting time ($O(n/p + \log^2 p)$) is equally as good as mapping an optimal EREW PRAM merge algorithm onto a hypercube or butterfly. It is an open question of whether the step complexity of sorting can be improved without effecting the work.

Machine Model	Time
CREW PRAM	$O(w/p + s \log p)$
CRCW PRAM	$O(w/p + s(\log\log p)^3)$
CRCW PRAM (randomized)	$O(w/p + s \log^* p)$
Butterfly (randomized)	$O(w/p + s \log p)$
Hypercube (randomized)	$O(w/p + s \log p)$

Figure 1: The mapping of Work (w) and Steps (s) in the proposed model (the A-PAL) to running time on various machine models. The number of processors on the machine is p. For the randomized algorithms the running times are high-probability bounds (*i.e.*, they will run within the specified time with very high probability). All the results assume that the number of independent variable names in a program is constant, as will be discussed in Section 4.

We chose applicative-order evaluation over normal-order evaluation because of ambiguities in defining a formal model based on normal-order evaluation. The problem is that normal-order evaluation can have wide range of implementations, such as call-by-name, call-by-need, and call-by speculation [16], and these implementations would have very different complexity models. The first two, call-by-need and call-by-name, actually offer no parallelism. Call-by-speculation offers plenty of parallelism but does the the same amount of work as applicative-order semantics. In particular, a model based on call-by-speculation would give the same asymptotic work bounds as our model, although it might be possible to improve some step bounds. Most implementations of lazy languages suggested in the literature sit somewhere between call-by-need and call-by-speculation. Typically some heuristic or strictness analysis is used to decide when to use call-by-speculation instead of call-by-need, and there is some way to garbage collect speculative computations that are never needed. In these implementations a complexity model would depend critically on what heuristics are used or how good the strictness analysis is. An interesting line of future work would be to formally compare implementation using their complexity models. For example it should be possible to show that one heuristic always gets as much parallelism as another without increasing the work.

We note that one inconvenience with our model is the need to keep track of how many variable names are needed. In particular our simulation bounds need to include the logarithm of the number of independent variables (v_e) in order to account for variable lookup. Fortunately it is straightforward to show that the number of variables for algorithms, such as sorting, is independent of the size of the input, so that v_e does not effect the asymptotic bounds. Another choice would be to restrict the λ-calculus to only allow a constant number of variables. This, however, would require that we chose a particular constant and then show how to convert programs with more variables into this fixed constant number.

Organization of the Paper

The paper is organized as follows. Section 2 describes the model and Section 3 describes an extended language with conditionals, recursion, data-types and local variables and shows that it is equivalent within constant factors to the base model. Sections 4 and 5 relate the model to various machine models. Section 6 gives algorithms for sorting and merging. Section 7 discusses related work.

2 The PAL Model

Our model is based on the untyped λ-calculus with an applicative order operational semantics augmented with complexity measures. We chose the λ-calculus rather than a specific language since its simplicity makes the simulation results in Section 4 much cleaner, and many features of modern languages (*e.g.*, data-types, conditionals, recursion, and local variables) can be simulated with constant overhead (Section 3), therefore not affecting asymptotic performance.

The parallelism in our model arises from evaluating the function and argument simultaneously and is specified by the definitions of the complexity measures. These are *work*, the total number of operations executed, and *steps*, analogous to depth in a circuit model. There is no notion of processors in the model, and in many ways the model more closely resembles circuit models than machine models. For the sake of practicality, we also consider an extension to the λ-calculus that adds a set of arithmetic constants (the integers along with some integer operators). This extension can be simulated on the pure model with costs polylogarithmic in the integer range. We will henceforth refer to the pure version as the *parallel applicative λ-calculus* (PAL) model and the extended version as the Arithmetic-PAL (A-PAL) model.

The abstract syntax of the model is

$$e \in \textit{Expressions} \quad ::= \quad c \mid x \mid \lambda x.e \mid e_1\ e_2$$

where the meta-variable c ranges over a set of constants. For the PAL model this set is empty, and for the A-PAL model it includes arithmetic constants.

We define the semantics of the language in terms of an evaluation relation. Each of the languages used in this paper is deterministic, so each of their evaluation relations will be functions. The possible values resulting from evaluation of a PAL expression are defined by

$$v \in \textit{Values} \quad ::= \quad c \mid cl(E, x, e)$$

A *closure* $cl(E, x, e)$ represents a function and denotes the value of a λ expression. Its first component is an *environment*, which is a finite mapping from variables to values. The empty environment is denoted by $[]$, and the extension of an environment with a variable and associated value is denoted by $E[x \mapsto v]$, where x may already be in E. If E has a binding for x, the associated value is denoted by $E(x)$.

Since we are using applicative order semantics and there are no side-effects in this model, the function and argument can be evaluated in parallel. This is the only form of parallelism we consider in this paper, and a goal of the paper is to demonstrate that this is a reasonably powerful model of parallel computation. To generate useful simulation results on machine models with bounded parallelism, it is important to keep track of the total work taken by a computation as well as the parallel depth of the computation. We therefore track two measures: the work complexity is the total number of reductions to evaluate the expression, and the step complexity is the time for evaluation assuming that e_1 and e_2 are always evaluated in parallel.

We formalize these complexities in terms of a *profiling semantics* for the language [33, 34]. In such a semantics, evaluating an expression always returns cost measures as well as the resulting value. Our profiling semantics is an extension of the standard environment-based operational semantics of the applicative order λ-calculus. The judgment $E \vdash e \xrightarrow{\lambda} v; s, w$ reads as "In the environment E, the expression e evaluates to value v in s steps and w work." When evaluating a program, we start with an empty environment. Our profiling semantics is defined by the rules in Figure 2.

Constants, λ-expressions, and variables evaluate in constant steps and work. As usual, constants evaluate to themselves, λ-expressions evaluate to closures, and the value of variables is determined by the current environment.

The APP and APPC rules define the application of user-defined and constant functions, respectively, where the meaning of a constant function application is given by the partial function δ. Parallel execution

$$E \vdash c \xrightarrow{\lambda} c; 1, 1 \qquad \text{(CONST)}$$

$$E \vdash \lambda x.e \xrightarrow{\lambda} cl(E, x, e); 1, 1 \qquad \text{(LAM)}$$

$$\frac{E(x) = v}{E \vdash x \xrightarrow{\lambda} v; 1, 1} \qquad \text{(VAR)}$$

$$\frac{E \vdash e_1 \xrightarrow{\lambda} cl(E', x, e'); s_1, w_1 \qquad E \vdash e_2 \xrightarrow{\lambda} v_2; s_2, w_2 \qquad E'[x \mapsto v_2] \vdash e' \xrightarrow{\lambda} v; s_3, w_3}{E \vdash e_1\ e_2 \xrightarrow{\lambda} v; \max(s_1, s_2) + s_3 + 2, w_1 + w_2 + w_3 + 2} \qquad \text{(APP)}$$

$$\frac{E \vdash e_1 \xrightarrow{\lambda} c; s_1, w_1 \qquad E \vdash e_2 \xrightarrow{\lambda} v_2; s_2, w_2 \qquad \delta(c, v_2) = v}{E \vdash e_1\ e_2 \xrightarrow{\lambda} v; \max(s_1, s_2) + 2, w_1 + w_2 + \delta_w(c, v_2)} \qquad \text{(APPC)}$$

Figure 2: The profiling semantics of the PAL model.

of a function and its argument is specified by combining their step complexity with max. Applying a constant function is assumed to take constant steps, a reasonable assumption for most constant functions, including those used here. But the amount of work depends on the function and its argument and is given by the function δ_w. The specific constant costs used here are selected to guarantee an exact correspondence between work and the number of reductions in an SECD machine (see Lemma 1).

Definition 1 *The* PAL model *is the λ-calculus with no constants and with the semantics defined by* $E \vdash e \xrightarrow{\lambda} v; s, w$.

Adding Constants to the PAL Model

We now extend the basic PAL model with arithmetic constants to obtain the Arithmetic-PAL model. These constants can be simulated on the pure version, but this would require non-constant overheads in both work and steps. The constants are

$$c \in \mathit{Constants} \quad ::= \quad \ldots \mid i \mid \mathbf{add} \mid \mathbf{add}_i \mid \mathbf{mul} \mid \mathbf{mul}_i \mid \mathbf{neg} \mid \mathbf{div2} \mid \mathbf{pos?}$$

where i ranges over the integers. The primitive functions are addition, multiplication, negation, division by two, and the test for positive integers. The choice of primitives is not important, but for the purpose of lower bounds proofs they should be incompressible [2], which ensures that certain kinds of data encoding schemes cannot asymptotically improve complexity bounds, *e.g.*, encoding arrays as integers. This is why general division has been omitted.

For syntactic simplicity, binary functions take one argument at a time, so that when applied to the first argument they return a new "curried" function that can be applied to the other argument. So the constants also include the results of applying the binary primitive functions to one argument, which are functions which expect the remaining argument. It is intended that these latter constants would not be used

$$\begin{array}{rclrcl}
\delta(\mathbf{add}, i) & = & \mathbf{add}_i & \delta(\mathbf{mul}, i) & = & \mathbf{mul}_i \\
\delta(\mathbf{add}_i, i') & = & i + i' & \delta(\mathbf{mul}_i, i') & = & i \times i' \\
\delta(\mathbf{neg}, i) & = & -i & \delta(\mathbf{div2}, i) & = & \lfloor i/2 \rfloor
\end{array}$$

$$\delta(\mathbf{pos?}, i) = \textit{if } i > 0 \textit{ then } cl([], x, \lambda y.x) \textit{ else } cl([], x, \lambda y.y)$$

$$\delta_w(c, i) = 1$$

Figure 3: The δ and δ_w functions for the A-PAL model.

in programs, but we have not fundamentally distinguished them from the other constants for the sake of simplicity.

The δ and δ_w functions for these constants are given in Figure 3. The two closures in the δ-rule for **pos?** are standard encodings for the booleans and can be used to encode conditionals as in Section 3. Applying each of these constants requires constant work.

Definition 2 *The* A-PAL model *is the λ-calculus with the constants* **add**, **mul**, **neg**, **div2**, *and* **pos?** *and with the semantics defined by* $E \vdash e \xrightarrow{\lambda} v; s, w$.

3 Extending the A-PAL Language

The λ-calculus by itself is too cumbersome a language for practical usage, but it does form the core of languges such as Lisp and ML. In this Section we define the μML model using the primary language constructs of these languages and show that this model can be translated to the A-PAL model with only constant overheads and adding only a constant number of variables. This implies that the simpler A-PAL model is sufficient for proving asymptotic complexity results in the μML model.

The μML model adds pairing, lists, booleans and conditionals, local variables, and explicit recursion to the PAL model. It also has more primitives and a syntax based on that of Standard ML. Its syntax is defined by

$$\begin{array}{rcl}
c \in \textit{Constants} & ::= & i \mid \texttt{+} \mid +_i \mid \texttt{-} \mid -_i \mid \texttt{*} \mid *_i \mid \texttt{/2} \mid \texttt{false} \mid \texttt{true} \mid \\
& & \texttt{=} \mid \texttt{>} \mid >_i \mid \texttt{not} \mid \texttt{nil} \mid \texttt{nil?} \mid \\
& & \texttt{cons} \mid \texttt{cons}_v \mid \texttt{hd} \mid \texttt{tl} \mid \texttt{fst} \mid \texttt{snd} \\
e \in \textit{Expressions} & ::= & c \mid x \mid (e_1, e_2) \mid \texttt{fn } x \texttt{ => } e \mid e_1\ e_2 \mid \\
& & \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \mid \texttt{letrec } x\ y \texttt{ = } e_1 \texttt{ in } e_2
\end{array}$$

A `let` expression defines the local variable x within e_2 and gives it the value of e_1. Similarly, a `letrec` expression defines the function x (with argument y) within e_2 and gives it the value of e_1. However, this also defines x within e_1, so its definition may be recursive.

The values of the language contain constants, cons-pairs, pairs, and closures. A special kind of closure is used for recursive functions in order to avoid using recursive environments:

$$v \in \textit{Values} \quad ::= \quad c \mid \langle v_1, v_2 \rangle \mid (v_1, v_2) \mid Cl(E, x, e) \mid ClR(E, x, e, y)$$

The profiling semantics are defined by the relation $E \vdash e \xrightarrow{ML} v; s, w$, which reads "In the environment E, the expression e evaluates to v in s steps and w work." It is defined by the rules given in Figure 4, using the δ and δ_work definition given in Figure 5.

$$E \vdash c \xrightarrow{ML} c; 1, 1 \qquad \text{(CONST)}$$

$$E \vdash \texttt{fn}\ x \texttt{=>}\ e \xrightarrow{ML} Cl(E, x, e); 1, 1 \qquad \text{(LAM)}$$

$$\frac{E(x) = v}{E \vdash x \xrightarrow{ML} v; 1, 1} \qquad \text{(VAR)}$$

$$\frac{E \vdash e_1 \xrightarrow{ML} Cl(E', x, e'); s_1, w_1 \qquad E \vdash e_2 \xrightarrow{ML} v_2; s_2, w_2 \qquad E'[x \mapsto v_2] \vdash e' \xrightarrow{ML} v; s_3, w_3}{E \vdash e_1\ e_2 \xrightarrow{ML} v; \max(s_1, s_2) + s_3 + 1, w_1 + w_2 + w_3 + 1} \qquad \text{(APP)}$$

$$\frac{E \vdash e_1 \xrightarrow{ML} c; s_1, w_1 \qquad E \vdash e_2 \xrightarrow{ML} v_2; s_2, w_2 \qquad \delta(c, v_2) = v}{E \vdash e_1\ e_2 \xrightarrow{ML} v; \max(s_1, s_2) + 1, w_1 + w_2 + \delta_w(c, v_2) + 1} \qquad \text{(APPC)}$$

$$\frac{E \vdash e_1 \xrightarrow{ML} v_1; s_1, w_1 \qquad E \vdash e_2 \xrightarrow{ML} v_2; s_2, w_2}{E \vdash (e_1, e_2) \xrightarrow{ML} (v_1, v_2); \max(s_1, s_2) + 1, w_1 + w_2 + 1} \qquad \text{(PAIR)}$$

$$\frac{E \vdash e_1 \xrightarrow{ML} \texttt{true}; s_1, w_1 \qquad E \vdash e_2 \xrightarrow{ML} v; s_2, w_2}{E \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \xrightarrow{ML} v; s_1 + s_2 + 1, w_1 + w_2 + 1} \qquad \text{(IFT)}$$

$$\frac{E \vdash e_1 \xrightarrow{ML} \texttt{false}; s_1, w_1 \qquad E \vdash e_3 \xrightarrow{ML} v; s_3, w_3}{E \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \xrightarrow{ML} v; s_1 + s_3 + 1, w_1 + w_3 + 1} \qquad \text{(IFF)}$$

$$\frac{E \vdash e_1 \xrightarrow{ML} v_1; s_1, w_1 \qquad E[x \mapsto v_1] \vdash e_2 \xrightarrow{ML} v_2; s_2, w_2}{E \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \xrightarrow{ML} v_2; s_1 + s_2 + 1, w_1 + w_2 + 1} \qquad \text{(LET)}$$

$$\frac{E[x \mapsto ClR(E, x, e_1, y)] \vdash e_2 \xrightarrow{ML} v_2; s, w}{E \vdash \texttt{letrec}\ x\ y = e_1\ \texttt{in}\ e_2 \xrightarrow{ML} v_2; s + 1, w + 1} \qquad \text{(LETREC)}$$

Figure 4: The profiling semantics of the μML model.

$$\delta(c,i) = c_i, \text{ if } c \in \{+,-,*,>,=\}$$

$$\begin{array}{rclrcl}
\delta(+_i,i') & = & i+i' & \delta(-_i,i') & = & i-i' \\
\delta(*_i,i') & = & i*i' & \delta(/2,i) & = & \lfloor i/2 \rfloor \\
\delta(>_i,i') & = & \textit{if } i > i' \textit{ then } \texttt{true} & \delta(=_i,i') & = & \textit{if } i = i' \textit{ then } \texttt{true} \\
 & & \textit{else } \texttt{false} & & & \textit{else } \texttt{false} \\
\delta(\texttt{not},\texttt{true}) & = & \texttt{false} & \delta(\texttt{not},\texttt{false}) & = & \texttt{true} \\
\delta(\texttt{nil?},\texttt{nil}) & = & \texttt{true} & \delta(\texttt{nil?},(v_1,v_2)) & = & \texttt{false} \\
\delta(\texttt{cons},v) & = & \texttt{cons}_v & \delta(\texttt{cons}_v,v') & = & \langle v,v' \rangle
\end{array}$$

$$\delta_w(c,v) = 1$$

Figure 5: The δ and δ_w functions for the μML model.

In order to relate the μML model to the PAL model, we define a translation function T on expressions, values, and environments. Figure 6 defines T on expressions and values, and this extends to environments in a point-wise manner.

The following theorem show that the μML model can be simulated by the simpler PAL model with only constant overheads. Thus, algorithms in the two models have the same asymptotic complexity bounds.

Theorem 1 *There exist k_S and k_W such that if $E \vdash e \xrightarrow{ML} v; s, w$ then $T[\![E]\!] \vdash T[\![e]\!] \xrightarrow{\lambda} T[\![v]\!]; s', w'$ such that $s' \leq k_S * s$, and $w' \leq k_W * w$.*

Proof: With the given definition of the A-PAL and μML models, suitable constants are $k_S = 12$ and $k_W = 16$. The values result from the complexity of the translation, particularly the `letrec` case, and the constants used in the A-PAL and μML model definitions. The proof has many cases, and we look at a representative few.

If e is a constant or an abstraction, then it is clear that the evaluation relations preserve the translation. Also, the given relations on the costs hold since $s = s' = 1$, and $w = w' = 1$.

If $e = (e_1, e_2)$ then it is again clear that evaluation preserves the translation. To show that the given relations on the costs hold, define the costs of evaluating e_i as s_i and w_i. Then $s = \max(s_1, s_2) + 1$, $w = w_1 + w_2 + 1$, $s' = \max(s'_1 + 3, s'_2) + 3$, and $w' = w'_1 + w'_2 + 7$. The `if`- and `let`-expression cases are similar.

The application case is also similar, except that we must look at the various possibilities of the value of the function.

The most complicated case is that of `letrec`-expressions. Without working through the whole derivation, note that the translation of a recursive closure is the result of evaluating $Y\ (\lambda x.\lambda y.T[\![e]\!])$ in the environment E. The overhead of this translation can be divided into two sources. First, there is overhead that is independent of the specific subexpressions e_1 and e_2, which is dominated by the evaluation of Y. This overhead shows that $k_S \geq 12$, and $k_W \geq 16$. Second, there is overhead for each application of the function x, since this involves unrolling the recursion. This overhead shows that $k_S \geq 9$, and $k_W \geq 12$. □

In addition, the translation T introduces only a constant number of variables, as shown by the following theorem. Together with the previous theorem, this shows that algorithms in the two models have the same asymtotic complexity bounds when mapped only models such as the RAM and PRAM.

Theorem 2 *There exists a constant k such that if $T[\![e]\!] = e'$, then $v_e \leq k + v_{e'}$.*

Expressions:

$$
\begin{array}{rcl}
T[\![x]\!] & = & x \\
T[\![+]\!] & = & \textbf{add} \\
T[\![-]\!] & = & \textbf{neg} \\
T[\![\texttt{true}]\!] & = & \lambda x'.\lambda y'.x' \\
T[\![\texttt{false}]\!] & = & \lambda x'.\lambda y'.y' \\
T[\![\texttt{nil}]\!] & = & \lambda x'.x'\,0\,0 \\
T[\![\texttt{nil?}]\!] & = & \lambda x'.0?\,(x'\,T[\![\texttt{true}]\!]) \\
T[\![\texttt{hd}]\!] & = & \lambda x'.T[\![\texttt{fst}]\!](T[\![\texttt{snd}]\!]\,x') \\
T[\![\texttt{tl}]\!] & = & \lambda x'.T[\![\texttt{snd}]\!](T[\![\texttt{snd}]\!]\,x') \\
T[\![\texttt{fst}]\!] & = & \lambda x'.x'\,(\lambda y'.\lambda z'.y') \\
T[\![\texttt{snd}]\!] & = & \lambda x'.x'\,(\lambda y'.\lambda z'.z')
\end{array}
\qquad
\begin{array}{rcl}
T[\![i]\!] & = & i \\
T[\![*]\!] & = & \textbf{mul} \\
T[\![/2]\!] & = & \textbf{div2} \\
T[\![\texttt{not}]\!] & = & \lambda x'.x'\,T[\![\texttt{false}]\!]\,T[\![\texttt{true}]\!] \\
 & & \\
T[\![=]\!] & = & \lambda x'.\lambda y'.0?\,(\textbf{add}\ x'\ (\textbf{neg}\ y')) \\
T[\![>]\!] & = & \lambda x'.\lambda y'.\textbf{pos?}\ (\textbf{add}\ x'\ (\textbf{neg}\ y')) \\
T[\![\texttt{cons}]\!] & = & \lambda x'.\lambda y'.\lambda z'.z'\ 1\ (\lambda z'.z'\ x'\ y') \\
T[\![(e_1, e_2)]\!] & = & (\lambda x'.\lambda y'.\lambda z'.z'\ x'\ y')\,T[\![e_1]\!]\,T[\![e_2]\!] \\
T[\![\texttt{fn}\ x\ \texttt{=>}\ e]\!] & = & \lambda x.T[\![e]\!] \\
T[\![e_1\ e_2]\!] & = & T[\![e_1]\!]\,T[\![e_2]\!]
\end{array}
$$

$$
\begin{array}{rcl}
T[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] & = & T[\![e_1]\!]\,(\lambda x'.T[\![e_2]\!])\,(\lambda x'.T[\![e_3]\!])\,0 \\
T[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!] & = & (\lambda x.T[\![e_2]\!])\,T[\![e_1]\!] \\
T[\![\texttt{letrec}\ x\ y = e_1\ \texttt{in}\ e_2]\!] & = & (\lambda x.T[\![e_2]\!])\,(Y\ (\lambda x.\lambda y.T[\![e_1]\!]))
\end{array}
$$

Values:

$$
\begin{array}{rcl}
T[\![+]\!] & = & \textbf{add} \\
T[\![-]\!] & = & \textbf{neg} \\
T[\![\texttt{true}]\!] & = & cl([\,], x', \lambda y'.x') \\
T[\![\texttt{false}]\!] & = & cl([\,], x', \lambda y'.y') \\
T[\![\texttt{nil}]\!] & = & cl([\,], x', x'\,0\,0) \\
T[\![\texttt{nil?}]\!] & = & cl([\,], x', 0?\,(x'\,T[\![\texttt{true}]\!])) \\
T[\![\texttt{hd}]\!] & = & cl([\,], x', T[\![\texttt{fst}]\!](T[\![\texttt{snd}]\!]\,x')) \\
T[\![\texttt{tl}]\!] & = & cl([\,], x', T[\![\texttt{snd}]\!](T[\![\texttt{snd}]\!]\,x')) \\
T[\![\texttt{fst}]\!] & = & cl([\,], x', x'\,(\lambda y'.\lambda z'.y')) \\
T[\![\texttt{snd}]\!] & = & cl([\,], x', x'\,(\lambda y'.\lambda z'.z'))
\end{array}
\qquad
\begin{array}{rcl}
T[\![*]\!] & = & \textbf{mul} \\
T[\![/2]\!] & = & \textbf{div2} \\
T[\![\texttt{not}]\!] & = & cl([\,], x', x'\,T[\![\texttt{false}]\!]\,T[\![\texttt{true}]\!]) \\
 & & \\
T[\![=]\!] & = & cl([\,], x', \lambda y'.0?\,(\textbf{add}\ x'\ (\textbf{neg}\ y'))) \\
T[\![=_i]\!] & = & cl([x' \mapsto i], y', 0?\,(\textbf{add}\ x'\ (\textbf{neg}\ y'))) \\
T[\![>]\!] & = & cl([\,], x', \lambda y'.\textbf{pos?}\ (\textbf{add}\ x'\ (\textbf{neg}\ y'))) \\
T[\![>_i]\!] & = & cl([x' \mapsto i], y', \textbf{pos?}\ (\textbf{add}\ x'(\textbf{neg}\ y'))) \\
T[\![\texttt{cons}]\!] & = & cl([\,], x', \lambda y'.\lambda z'.z'\ 1\ (\lambda z'.z'\ x'\ y')) \\
T[\![\texttt{cons}_v]\!] & = & cl([x' \mapsto T[\![v]\!]], y', \lambda z'.z'\ x'\ y')
\end{array}
$$

$$
\begin{array}{rcl}
T[\![\langle v_1, v_2\rangle]\!] & = & cl([x' \mapsto T[\![v_1]\!], y' \mapsto T[\![v_2]\!]], z', z'\ 1\ (\lambda z'.z'\ x'\ y')) \\
T[\![(v_1, v_2)]\!] & = & cl([x' \mapsto T[\![v_1]\!], y' \mapsto T[\![v_2]\!]], z', z'\ x'\ y') \\
T[\![Cl(E, x, e)]\!] & = & cl(T[\![E]\!], x, T[\![e]\!]) \\
T[\![ClR(E, x, e, y)]\!] & = & cl(T[\![E]\!][x \mapsto cl(E'[y' \mapsto cl(E', f, x'\ (\lambda z'.y'\ y'\ z'))], z', y'\ y'\ z')], y, T[\![e]\!]) \\
 & & \text{where } E' = T[\![E]\!][x' \mapsto Cl(T[\![E]\!], x, \lambda y.T[\![e]\!])]
\end{array}
$$

using the abbreviations

$$
\begin{array}{rcl}
0? & = & (\lambda x'.(\textbf{pos?}\ x')\,T[\![\texttt{false}]\!]\,((\textbf{pos?}\ (\textbf{neg}\ x'))\,T[\![\texttt{false}]\!]\,T[\![\texttt{true}]\!])) \\
Y & = & \lambda x'.(\lambda y'.x'\ (\lambda z'.y'\ y'\ z'))\,(\lambda y'.x'\ (\lambda z'.y'\ y'\ z'))
\end{array}
$$

Figure 6: The translation function T from the μML model to the PAL model. The variable name x' is assumed to be distict from all others used in the expression being translated.

Proof: The translation T involves a fixed number of variables, which fall into two classes. First, x and y are used as metavariables representing variables in the original expression e. Thus any variable occurring in e is also in its translation e'. Second, the translation introduces at most $k = 3$ variables, x', y', and z', which may be independent of those in e. □

Many other language extensions would add only constant overheads. In particular, recursive datatype definitions and the associated pattern matching such as that in Standard ML can be defined in the same way that lists are defined here. Each constructor (`nil`, `cons`) tags its data, each destructor (`hd`, `tl`) selects the appropriate component, and each mutator (`nil?`) tests for the appropriate tag. Pattern matching is built upon such mutators. Such datatype definition and pattern matching is assumed in Section 6.

4 Simulating the A-PAL on Various Machines

In this section we prove simulation bounds for simulating the A-PAL model (or PAL) on various machine models. We first describe the simulation on a serial RAM and then extend this for the simulation on a PRAM, hypercube and butterfly network. To simulate the A-PAL on the RAM, we use a variant of the SECD machine [21, 27] as an intermediate step. We first show how the work complexity of an A-PAL program is related to the number of state transitions of the SECD machine and then show that each transition can be implemented within given bounds. For the parallel simulations of the A-PAL, we introduce a parallel variant of the SECD machine, the Parallel ECD (P-ECD) machine. The basic idea of the P-ECD machine is that it keeps a set of substates that can be evaluated in parallel. A state transition causes each substate to convert into either 0, 1, or 2 new substates, so the number of substates will vary over the computation. We show that the work complexity of a program is exactly equal to the total number of substates processed and that the step complexity is exactly equal to the number of steps taken by the P-ECD machine. We then show using an appropriate scheduling how this can be mapped onto various machines with a fixed number of processors.

We now briefly review the SECD machine. It is a state machine with transition function $\stackrel{S}{\Longrightarrow}$, where states consist of a data stack S of values, an environment E, a control list C of expressions or the symbol @ (*apply*), and a "dump" D which is a list of (S, E, C) triples used as a control stack to return from function calls. To evaluate an expression e, the machine starts in the state $(nil, nil, [e], nil)$. It halts when S is a singleton and both C and D are *nil*, with the result being the singleton value in S. The state transition function is given in Figure 7.

Now we define the cost of the SECD transitions and relate the work cost in the A-PAL model to that of the SECD machine. The cost of each SECD transition is the constant 1, except for prim-calls which have cost $\delta_w(c, v)$. Based on the SECD machine, calculating the mapping between work in A-PAL model and time on a RAM can be split into determining the mapping of work on the A-PAL to the cost in the SECD machine and then relating this cost that in the RAM. This includes determining the maximum RAM time taken by each non-prim-call transition.

Lemma 1 *If* $[] \vdash e \xrightarrow{\lambda} v; s, w$, *then the SECD machine evaluates* e *to* v *with* w *cost.*

Proof: First, we generalize the lemma to the intermediate states of the SECD machine: *If* $E \vdash e \xrightarrow{\lambda} v; s, w$, *then the transition sequence* $(S, E, e :: C, D) \stackrel{S}{\Longrightarrow}^{*} (v :: S, E, C, D)$ *has cost* w. Then, the proof is by structural induction on the A-PAL evaluation derivation, with a case analysis on the last rule used in this derivation.

S	E	C	D		S', E', C', D'	
$S,$	$E,$	$c :: C,$	D	$\stackrel{S}{\Longrightarrow}$	$c :: S, E, C, D$	constant
$S,$	$E,$	$(\lambda x.e) :: C,$	D	$\stackrel{S}{\Longrightarrow}$	$cl(E, x, e) :: S, E, C, D$	lambda
$S,$	$E,$	$x :: C,$	D	$\stackrel{S}{\Longrightarrow}$	$E(x) :: S, E, C, D$	variable
$S,$	$E,$	$(e_1\ e_2) :: C,$	D	$\stackrel{S}{\Longrightarrow}$	$S, E, e_2 :: e_1 :: @ :: C, D$	apply
$cl(E', x, e) :: v :: S,$	$E,$	$@ :: C,$	D	$\stackrel{S}{\Longrightarrow}$	$nil, E'[x \mapsto v], [e], (S, E, C) :: D$	func-call
$c :: v :: S,$	$E,$	$@ :: C,$	D	$\stackrel{S}{\Longrightarrow}$	$\delta(c, v) :: S, E, C, D$	prim-call
$v :: S,$	$E,$	$nil,$	$(S', E', C') :: D$	$\stackrel{S}{\Longrightarrow}$	$v :: S', E', C', D$	return

Figure 7: The transitions of the SECD machine. The notation $a :: b$ denotes the element a added to the front of the list b.

CONST, LAM, or VAR: The SECD machine requires one constant, lambda, or variable transition, and $w = 1$. The resulting value is the same in both the A-PAL and SECD machine by simple inspection of the corresponding rules.

APP: By induction and instantiating the intermediate states as needed, we have that

$$\begin{aligned}
(S, E, e_2 :: e_1 :: @ :: C, D) &\stackrel{S}{\Longrightarrow}{}^{*} (v_2 :: S, E, e_1 :: @ :: C, D) \\
(v_2 :: S, E, e_1 :: @ :: C, D) &\stackrel{S}{\Longrightarrow}{}^{*} (cl(E', x, e') :: v_2 :: S, E, @ :: C, D) \\
(nil, E'[x \mapsto v_2], [e'], (S, E, C) :: D) &\stackrel{S}{\Longrightarrow}{}^{*} ([v], E'[x \mapsto v_2], nil, (S, E, C) :: D)
\end{aligned}$$

and that these transition sequences are of cost w_2, w_1, and w_3, respectively. To complete the desired sequence of transitions, we add one func-call transition between the last two previous sequences and one return transition at the end. Thus, the SECD transition sequence is of cost $w = w_1 + w_2 + w_3 + 2$.

APPC: By induction and instantiating the intermediate states to as needed, we have that

$$\begin{aligned}
(S, E, e_2 :: e_1 :: @ :: C, D) &\stackrel{S}{\Longrightarrow}{}^{*} (v_2 :: S, E, e_1 :: @ :: C, D) \\
(v_2 :: S, E, e_1 :: @ :: C, D) &\stackrel{S}{\Longrightarrow}{}^{*} (c :: v_2 :: S, E, @ :: C, D)
\end{aligned}$$

and that these sequences are of cost w_2 and w_1, respectively. The sequence of transitions is finished by a prim-call, for a total cost of $w = w_1 + w_2 + \delta_w(c, v_2)$.

□

In the following lemma, v_e is the logarithm of the number of independent variable names in an expression e. In the worst case this is equal to the number of λ-expressions since each could have its own variable name, but we assume that names are shared among λs where it does not cause a conflict. In practice v_e is a small constant that is independent of the data size—it is easy to share names in all common data representations. In general, however, it is possible to define data representations in which v_e is a function of the data size, so it is important to keep track of it.

Lemma 2 *Each non-prim-call step of an SECD machine on an expression e starting with an empty environment can be simulated on a RAM in no more than kv_e time, for some constant k.*

Proof outline: All transitions except for environment lookup ($E(x)$) and environment extension ($E[x \mapsto v]$) can be implemented with simple list manipulations and take constant time. If the environment is implemented as a balanced tree, then the environment lookup and extension can be implemented in time logarithmic in the number of variable names in the environment. This assumes there is a total order on the variable names, and is a little trickier than expected since environment modification requires making a copy of the old environment (it cannot be side effected). When evaluating an expression e with an initially empty environment the number of variables names in the environment can never exceed v_e. □

We note that Lemma 2 is also true for a pointer machine [20, 38, 2] since the simulation does not require any random access.

Theorem 3 *If* $[] \vdash e \xrightarrow{\lambda} v; s, w$ *and a RAM can calculate each primitive call* $\delta(c, v)$ *in* $kv_e\delta_w(c, v)$ *time, then* v *can be calculated from* e *on a RAM in no more than* $kv_e w$ *time, for some constant* k.

Proof: Follows from Lemmas 1 and 2. □

For the parallel simulation we introduce the P-ECD machine. Again the simulation can be split into relating the complexity of the A-PAL to the number of state transitions of the P-ECD, and then we can bound the time to execute each transition and various parallel machines.

The P-ECD machine consists of a controlling processor and a set of slave processors. The state of the machine is a pair (Q, M). The first component is an array of substates, each similar to a SECD state, but without the stack:

$$Q = [(E_1, C_1, D_1), (E_2, C_2, D_2), \ldots, (E_n, C_n, D_n)].$$

The second is an array of optional partial results, thus taking the place of the stack:

$$M = [V_1, V_2, \ldots, V_m],$$

where each V_i is either has zero (*noval*) or one (*val*(v)) partial result.

Each step of the P-ECD machine transforms the current state. To evaluate an expression e, the machine starts in the state $([(nil, e, nil)], [])$ and exits with the value of e. A step consists of first allocating the substates in Q to the slave processors; executing a substate transition on each slave, each returning 0, 1 or 2 new substates or exiting with a value; and accumulating these substates as the new array of substates. The entire computation finishes when one slave exits. It is impossible for more than one processor to exit or for there to be no new substates unless the computation is exiting.

The substate transition executed on each processor works in three substeps, *eval*, *valf*, and *vala*, as defined in Figure 8. The *eval* substep creates intermediate results of evaluation which are processed by *valf* and *vala* into substates. This processing includes coordinating the values obtained from evaluating functions and arguments, and so the processors must synchronize between these latter substeps. Array M can be side-effected by the substeps: *eval* can extend the array, and *valf* and *vala* can update its contents.

We now argue informally why the machine works. The interesting transitions are *eval* on applications $(e_1\ e_2)$ and the non-identity *valf* and *vala* transitions. This *eval* transition creates two new substates to evaluate the function and argument. The index i added to the dump D is guaranteed to be independent for each substate processed (*e.g.*, the processor ID plus the number of substates processed in previous steps) and is used as an index into M. Whichever calculation completes first writes its result into $M[i]$ and returns no substates. Whenever the second calculation completes, it reads the result from $M[i]$ and initiates the application of v_1 to v_2. In the case that the two branches complete on the same step, we guarantee that they both do not believe that the other is still running by synchronizing between the *valf* and *vala* phases. (With an atomic test-and-set, synchronizing could be avoided.)

$E,\ c,$	$D \overset{eval}{\Longrightarrow}$	$\mathbf{res}(c, D)$	constant
$E,\ \lambda x.e,$	$D \overset{eval}{\Longrightarrow}$	$\mathbf{res}(cl(E, x, e), D)$	lambda
$E,\ x,$	$D \overset{eval}{\Longrightarrow}$	$\mathbf{res}(E(x), D)$	variable
$E,\ e_1\ e_2,$	$D \overset{eval}{\Longrightarrow}$	$M[i] := noval;$ $\mathbf{2S}((E, e_1, (E, i, \mathbf{fn}) :: D), (E, e_2, (E, i, \mathbf{arg}) :: D))$ where i is new	apply
$E,\ @(cl(E, x, e), v),$	$D \overset{eval}{\Longrightarrow}$	$\mathbf{1S}(E[x \mapsto v], e, D)$	func-call
$E,\ @(c, v),$	$D \overset{eval}{\Longrightarrow}$	$\mathbf{res}(\delta(c, v), D)$	prim-call
$\mathbf{res}(v, nil)$	$\overset{valf}{\Longrightarrow}$	$\mathbf{Exit}(v)$	exit
$\mathbf{res}(v, (E, i, \mathbf{fn}) :: D)$	$\overset{valf}{\Longrightarrow}$	*if hasval*$(M[i])$ *then* $\mathbf{1S}(E, @(v,$ *valof*$(M[i])), D)$ *else* $(M[i] :=$ *val*$(v); \mathbf{0S})$	left-return
$\mathbf{res}(v, (E, i, \mathbf{arg}) :: D)$	$\overset{vala}{\Longrightarrow}$	*if hasval*$(M[i])$ *then* $\mathbf{1S}(E, @($*valof*$(M[i], v)), D)$ *else* $(M[i] :=$ *val*$(v); \mathbf{0S})$	right-return

Otherwise, *valf* and *vala* are identities.

Figure 8: Transitions on the substates of the P-ECD. The notation $M[i] := z$ denotes writing z into the i^{th} element of M. The $s1; s2$ notation signifies sequentially executing $s1$ and then $s2$.

An example P-ECD evaluation trace is in Figure 9. It shows the expressions in Q at the beginning of each step of evaluating (**add** (**add** 1 2) (**add** 3 4)).

Like in the SECD machine, the cost of each *eval* substep is 1. Furthermore, we assume in Lemma 4 and Theorem 4 that $\delta_w(c, v)$ is constant for each prim-call, as in the A-PAL model, in order to simplify descriptions and proofs. The proofs can be generalized to hold without this assumption.

Lemma 3 *For all expressions e, if there exists a value v such that* $[] \vdash e \xrightarrow{\lambda} v; s, w$*, then v is calculated from e using exactly s steps of a P-ECD machine. Furthermore, the P-ECD calculation processes exactly w states.*

Proof: We prove that the number of steps taken by the P-ECD machine is s by induction on the structure of the A-PAL evaluation derivation. The induction hypothesis is that if $E \vdash e \xrightarrow{\lambda} v; s, w$ and the P-ECD machine at step t is in a state (Q, M) such that substate (E, e, D) is in Q, then an instance of the *eval* substep of step $t + s - 1$ results in $\mathbf{res}(v, D)$.

CONST, LAM, or VAR: The current *eval* substep results in $\mathbf{res}(v, D)$. By the profiling semantics, $s = 1$, so the hypothesis is true.

APP: By the *eval* rules, two substates (E, e_1, D_1) and (E, e_2, D_2) are created after one step. By the induction hypothesis, e_1 completes after s_1 steps, and e_2 completes after s_2 steps. If the calculation for e_1 completes before the calculation for e_2 (*i.e.*, $s_1 < s_2$), then when e_2 completes, $(E, @(v_1, v_2), D)$ is in the array of substates at step $t + 1 + s_2$. Otherwise, when e_1 completes, $(E, @(v_1, v_2), D)$ is in the array of substates at step $t + 1 + s_1$. Therefore, $(E', @(cl(E, x, e), v2), D)$ is in the array of

Step	expressions in Q	$\|Q\|$
1	**(add (add** 1 2) **(add** 3 4))	1
2	**(add (add** 1 2)), **(add** 3 4)	2
3	**add**, **(add** 1 2), **add** 3, 4	4
4	**add** 1, 2, **add**, 3	4
5	**add**, 1, **@(add**,3)	3
6	**@(add**,1), **@(add**$_3$,4)	2
7	**@(add**$_1$,2)	1
8	**@(add**,3)	1
9	**@(add**$_3$,7)	1
	Work:	19

Figure 9: P-ECD example evaluation using the expression (**add** (**add** 1 2) (**add** 3 4)).

substates at step $t + 1 + \max(s_1, s_2)$. At the beginning of the next step, $t + 2 + \max(s_1, s_2)$, the substate $(E[x \mapsto v], e, D)$ is in the array of substates. By the induction hypothesis, an instance of the *eval* substep of step $(t + 2 + \max(s_1, s_2)) + s_3 - 1$ results in **res**(v, D). Since the profiling semantics shows that $s = 2 + \max(s_1, s_2) + s_3$, this gives the desired results.

APPC: The argument is the similar to the previous rule, except that at the beginning of step $t + 1 + \max(s_1, s_2)$ the substate $(E, @(c, v2), D)$ is in the array of substates, and an instance of the *eval* substep results in **res**(v, D).

Now we show that the cost of the calculation is not more than w. The proof is by induction on the A-PAL derivation.

CONST, LAM, or VAR: Exactly one P-ECD step is needed for each of these A-PAL rules, and this step has a cost of $w = 1$.

APP: By induction, the values of e_1, e_2, and e' are calculated in not more than w_1, w_2, and w_3 cost, respectively. In addition, one func-call *eval* substep, of cost 1, is taken prior to the evaluation of e'. Thus, the cost is less than $w = w_1 + w_2 + w_3 + 2$.

APPC: By induction, the values of e_1 and e_2 are calculated in not more than w_1 and w_2 cost, respectively. In addition, one prim-call *eval* substep, of cost $\delta_w(c, v_2)$, is taken to complete the evaluation of the application's value. Thus, the cost is not more than $w = w_1 + w_2 + \delta_w(c, v_2)$.

□

We now need to show how to simulate the P-ECD machine on a PRAM and butterfly network. For the butterfly we assume that for p processors we have $p \lg p$ switches and p memory banks, and that memory references can be pipelined through the switches. On such a machine each of p processor can access (read or write) n elements in $O(n + \log p)$ time with high probability [23, 28]. The $O(\log p)$ time is due to latency through the network. We also assume the butterfly network has simple integer adders in the switches, such that a prefix-sum computation can execute in $O(\log p)$ time. A separate prefix tree, such as on the CM-5, would also be adequate. For the hypercube we assume a multiport hypercube in which on each time step messages can cross all wires, and for which there are separate queues for each wire. This model is quite

similar to butterfly and has the same bounds for simulating shared memory. However, we do not need to assume that the switches have integer adders.

Lemma 4 *If each primitive call $\delta(c, v)$ can be calculated on one processor in constant time, then one step of the P-ECD machine with m states can be processed on a p processor machine within the following time bounds:*

$k \cdot v_e \cdot (\lceil m/p \rceil + \log p)$	*CREW PRAM*
$k \cdot v_e \cdot (\lceil m/p \rceil + (\log\log p)^3)$	*CRCW PRAM*
$k \cdot v_e \cdot (\lceil m/p \rceil + \log^* p)$	*randomized CRCW PRAM (w.h.p.)*
$k \cdot v_e \cdot (\lceil m/p \rceil + \log p)$	*randomized Butterfly (w.h.p.)*

Proof: For the simulation we keep the substates returned by each step in an array. If this substate array is of length n, each processor is responsible for n/p elements of the array (*i.e.*, processor i is responsible for the elements $[in/p, \ldots, (i+1)n/p - 1]$). We assume each processor knows its own processor number, so it can calculate a pointer to its section of the array. For the CREW and butterfly simulations the length of the array is exactly m, the number of substates. For the CRCW PRAM simulations the array can have holes in it that don't contain states, as explained below. These holes are marked, and we guarantee that the total length of the array is at most km for some constant k. This means that each processor is responsible for at most km/p elements.

The simulation of a step consists of the following substeps:

1. Locally evaluating the substates using the *eval* transition in Figure 8. This requires accessing shared memory for reading but requires no communication among the substates. Each transformed substate can be written back into the array location from which it was read.

2. Evaluating the *valf* and *vala* transitions. This requires a synchronization between the two transitions. Each processor first uses the *valf* transitions for all the substates for which it is responsible. The processors then synchronize, and then each processor uses the *vala* transitions.

3. Creating a new substate array for the next step. After the substep transitions, each array element contains zero, one, or two substates (**0S**, **1S**, or **2S**), and these must be distributed into the new array.

We need to show that each of these steps can be executed in the given bounds. The first step requires the time it takes to process n/p substates. The *eval* transition is similar to the *eval* for the serial SECD machine. The only real difference is the apply transition. Each of the other state transition require the v_e time that was required in the serial machine and can have at most v_e memory references. The apply transition can also be executed in these bounds since it just requires an additional memory write. We can generate the independent i's simply by using the array index for the substate added to an offset which gets reset on each round. None of the memory references require concurrent writes. The time for the first substep on the CREW and CRCW PRAM is therefore n/p. The time on the butterfly is $m/p + \lg p$ since the memory references require a $\lg p$ latency through the network. The second step can also be executed in the same bounds.

The third step requires generating a new substate array. Each transitioned substate of the old array contains zero, one, or two substates, which need to be distributed into a new array for the next step. For the CREW PRAM and butterfly this can be done by executing a prefix-sum on the number of new substates and using the result as an offset into the new array. In both cases for p processors the prefix sum and writing into the new array can run in $O(m/p + \log p)$ time. This will give a new array that is exactly the length of the number of new substates. On the CRCW PRAM the distribution into the new array can be done more efficiently using a solution to the *linear approximate compaction* problem [22]: given an array of n cells, m

of which contain an object, place the m objects in distinct cells of an array of size km for some constant k. The idea is to first allocate two new positions for each substate, mark the substates that will remain (neither for **0S**, one for **1S**, and both for **2S**) and then do an approximate compaction. Since the result array is a constant times larger than the total number of remaining states, we will maintain the invariant mentioned earlier. Gil, Matias, and Vishkin [13] have shown that the linear approximate compaction problem can be solved on a p processor CRCW PRAM (ARBITRARY) in $O(n/p + \log^* p)$ expected time (using a randomized solution). Hagerup [15] has shown that the problem can be solved deterministically in $O(n/p + (\log\log p)^3)$ time.

When we add the times for the three substeps, we get the stated bounds for each of the machines. □

Theorem 4 *If* $[] \vdash e \xrightarrow{\lambda} v; s, w$, *and each primitive call* $\delta(c, v)$ *can be calculated on one processor in constant time, then* v *can be calculated from* e *on a CREW PRAM with* p *processors within* $kv_e(w/p + s\log p)$ *time, for some constant* k. *Analogous results are true for the other models.*

Proof: The proof uses Brent's scheduling principle [7]. We prove it for the CREW PRAM, but the other proofs are almost identical. We assume that each step of the P-ECD processes w_i substates. We know from Lemma 3 that $\sum_{i=0}^{i<s} w_i = w$. We also know from Lemma 4 that it will take $k'v_e(\lceil w_i/p \rceil + \log p)$ to process step i (note that we have introduced k' so that it is not confused with the k in this theorem). The total time to process all states is then

$$\begin{aligned} T &= \sum_{i=0}^{i<s} k'v_e(\lceil w_i/p \rceil + \log p) \\ &< k'v_e \sum_{i=0}^{i<s} (w_i/p + 1 + \log p) \\ &< k'v_e((\sum_{i=0}^{i<s} w_i/p) + s(1 + \log p)) \\ &< k'v_e(w/p + s(1 + \log p)) \\ &< 2k'v_e(w/p + s\log p)) \\ &< kv_e(w/p + s\log p)) \end{aligned}$$

where we have set $k = 2k'$. □

5 Simulating a PRAM on an A-PAL

In this section we consider simulating a PRAM on an A-PAL. The simulation we use gives the same results for the EREW, CREW, and CRCW PRAM as well as for the multiprefix [29] and scan models [4]. The simulation is optimal in terms of work for all the PRAM variants since there is a lower bound of $O(\log M)$ work required for each random access (this is the same as for pointer machines [2]). Since we don't know how to do better for the weaker models, we will base our results on the most powerful model, the CRCW PRAM with unit time multiprefix sums (MP PRAM).

Theorem 5 *A program that runs in time* t *on a* p *processor MP PRAM using* m *memory can be simulated on the A-PAL model with* $s = k_S t \log m \log p$, *and* $w = k_W p \log m$, *for some constants* k_S *and* k_W.

Proof: We will simulate a PRAM based on state transitions on the state (C, M, P) where C is the code, M is the memory, and P is state for all the processors (*i.e.*, registers and program counter). We assume C, M, and P are stored as balanced binary trees and that $(p = |P|) \leq (m = |M|)$, and $|C| \leq m$. Each state transition corresponds to a step of the PRAM, and the processors will be strictly synchronous. Register-to-register instructions can be implemented with $s = O(\log p)$, and $w = O(p)$, and concurrent reads with $s = O(\log m)$, and $w = O(p \log m)$. This just requires traversing the appropriate trees. The writes are the only interesting instruction to implement, and can be implemented by sorting the write requests from the processors by address and then recursively splitting the requests at each node of the M tree as we intert them. Since we have p requests, the sort of the requests can be implemented in $s = O(\log^2 p)$, and $w = O(p \log p)$ as discussed in the next section. We assume the sorted requests, which we call the write-tree, start out balanced and are sorted from left to right in the tree. To implement a concurrent write or multiprefix, we combine nodes in the write-tree that have the same address. Since the addresses are sorted this can be done in $s = O(\log p)$ and $w = O(p)$.

We now consider the insertion of the sorted requests into the M tree. It will be based on a recursive routine `modify(`M`,`W`)` which takes a memory tree M with a range of addresses along with associated values and a write-tree W with locations to modify in the M tree along with new values. We assume all locations in W are contained in M. We also assume for M that the addresses and associated values are stored at the leaves, that the addresses are ordered from left-to-right, and that the internal nodes contain the value of the greatest address in the left branch. For W we keep the minimum and maximum addresses along with the write-tree such that we can access these in constant work and steps. To insert W into M we first check if M is a single node, in which case W must also be a single node and we simply modify the value and return. Otherwise we check if all the addresses in W go down just one of the branches of the M tree. If all addresses go down one branch we just call `modify` recursively on that branch of M with the same W and put the result back together with the other branch of M when the call returns. If W belongs on both branches of M, we split W based on the address stored at the node in M and call `modify` in parallel on the two children of M and the two split parts of W. This algorithm works since all addresses in the original write-tree will eventually find their way to the appropriate leaf of the M tree and modify that leaf.

We now consider the total work and steps required. The splitting of W into two trees based on a key can be implemented in $s = w = O(\log p)$ by just following down to the appropriate leaf splitting along the way (this is a simplified version of the `in_range` operation discussed in the next section). Since the M tree is of depth $\lg m$, the total step complexity is bound by $O(\log p \log m)$. To prove the bounds on the work, we observe that it cannot take more than $O(p \log p)$ work to split the tree into p pieces of size 1 since each split will take $O(\log p)$ work and there will be $p - 1$ of them. This means the total work done on splitting the original write-tree is bound by $O(p \log p)$. The only other work is the check at each node of the M tree of whether we have to split or send all values down to one or the other branches. The maximum work done for these checks is $O(p \log m)$ since there can be at most p separate chains (one per leaf of the write-tree) each which is at most as deep as the M tree ($O(\log m)$). The total work is therefore $O(p(\log p + \log m)) = O(p \log m)$. □

6 Bounds for Merging and Sorting

In this section we give algorithms for merging and sorting for the A-PAL model. It is easy to show lower bounds for both problems of $s = \lg n$, where n is the size of the data since it is only possible to fork at most two parallel calls on each step. The lower bounds for work are the same as the sequential lower bounds for the problems—$O(n)$ for merging and $O(n \lg n)$ for sorting.

We consider the problem of merging two ordered sequences. We give an algorithm with optimal

complexity $s = O(\log n)$, and $w = O(n)$, where n is the length of the result. The algorithm determines $n/\lg n$ splitters that partition the result exactly and uses these splitters to extract the appropriate subsequences of the two inputs, appending the results. Note that algorithms based on partitioning each input sequence into equal sized blocks, such as the PRAM algorithm of Shiloach and Vishkin [35], cannot be directly implemented efficiently on the A-PAL model. This is because it is hard without side-effects to do the patching between the two sequences. Also note that given a solution of the ranking problem (each element in a has its rank in b and vice versa), it remains nontrivial to solve the merging problem work efficiently in the A-PAL. In the PRAM models it is trivial because of the ability to use random access.

For our algorithm we store ordered sequences in a tree structure with all values kept at the leaves. Each internal node holds pointers to its two children, the size of the sequence (the number of leaves below it), and the maximum value of any leaf below it. The order of the sequence is given by the left-to-right traversal of the tree. We denote the depth of sequence a with $D(a)$. The algorithm uses the following subroutines:

`map (f, a)`
Takes a function f and a sequence $[a_0, a_1, \ldots, a_{n-1}]$ and returns $[f(a_0), f(a_1), \ldots, f(a_{n-1})]$. The complexity is $s = O(D(a) + \max_{i=0}^{i<n} s(f(a_i)))$, and $w = O(\sum_{i=0}^{i<n} w(f(a_i)))$.

`iseq (start, end, stride)`
Returns an integer sequence starting at *start*, up to but not including *end* with stride *stride*. The complexity is $s = O(\log l)$, and $w = O(l)$, where $l = (end - start)/stride$.

`in_range (a, v0, v1)`
Takes an ordered sequence $a = [a_0, a_1, \ldots, a_{n-1}]$ and returns an ordered subsequence of a with all elements such that $v_0 \leq a_i < v_1$. To implement it, we execute a binary-tree search for v_0 in a and drop the left branch whenever we take a right during the search. We then do a binary search on the result with v_1 and drop the right branch whenever we take a left. The code is shown in Appendix A. The work and step complexities are both $O(D(a))$, and the result is at most the same depth as the source.

`kth_smallest (k, a, b)`
Given two ordered sequences a and b, this returns the k^{th} smallest value from the combination of the two sequences. It is implemented using a dual binary search in which, on each step, we go down a branch from one of the two sequences. The code is shown in Appendix A, and its complexity is $s = w = O(D(a)+D(b))$.

`serial_merge (a, b)`
Serially merges the two ordered sequence and returns a balanced ordered sequence. $s = w = O(|a| + |b|)$.

Theorem 6 *Two ordered sequences a and b, each stored as a balanced tree, can be merged in the A-PAL model with complexities $s = O(\log n)$, and $w = O(n)$, where n is the size of the result. The result is returned as a balanced tree.*

Proof: The code for merging is given in Figure 10. The call to `iseq` returns a sequence of integers that evenly partition the result into $n/\lg n$ parts. The calls to `extract` then extract exactly $\lg n$ elements each, except for the last which might extract fewer. The complexity for each call to `extract` is $s = w = O(\log n)$ since that is the bound for each of the subcalls. The `flatten` instruction simply flattens the nested sequence into a sequence. Using the equation for the complexity of `map`, the total complexity is $s = O(\log n)$, and $w = O(n)$. □

We note that the total number of variables in the merge program is independent of the size of the input data such that v_e is constant. This matters when we map the program onto the various machine models.

```
/* a and b are the two input sequences stored as trees
   i is the start of the region to extract
   j is the end of the region to extract */
fun extract (s1,s2,i,j) =
let v1 = kth_smallest (i,s1,s2)
    v2 = kth_smallest (j,s1,s2)
in serial_merge(in_range (s1,v1,v2),in_range (s2,v1,v2))
end

fun parallel_merge (s1,s2) =
let n = + (size s1) (size s2)
    p = iseq (0,n,lg n)    /* Create the sequence 0, lg n, 2lg n, ... */
    b = map ((fn i => extract (s1,s2,i,+ i (lg n))),
             p)   /* Apply extract to each region of length lg n */
in flatten b
end
```

Figure 10: Code for merging.

Using the merge described above, it should be clear that mergesort can be implemented with $s = O(\log^2 n)$, and $w = O(n \log n)$. It is possible to sort in $s = O(\log n)$, and $w = O(n^2)$ by counting for each key how many of the other keys are less than it, or equal and to the left in the tree. This gives the rank position of each element in the final tree, which can then be use to select out the element that belongs at each position in the final tree. The question is remains, however, of whether can sort work efficiently with $s = o(\log^2 n)$? In the EREW PRAM, Cole's sort sorts in $O(\log n)$ time with n processors [8]. This algorithm cannot be used directly since it requires random access. Goodrich and Kosaraju showed how this bound could also be achieved in the EREW parallel pointer machine (PPM) [14]. It does not seem however that this algorithm can be modified to work in the A-PAL model either. The problem is that the algorithm requires side-effects (*e.g.*, doubly linked lists), which our model does not allow. We should also point out that in the PPM it is possible to create a DAG that emulates an AKS network and sorts in the same bounds. Again this seems unlikely for the A-PAL.

7 Related Work

Several researchers have used cost-augmented semantics for automatic time analysis of serial programs [3, 33, 34, 39]. This work was concerned with serial running time, and since they were primarily interested in automatically analyzing programs rather than defining complexity, they each altered the semantics of functions to simplify such analysis. Furthermore, none related their complexity models to more traditional machine models, although since the languages are serial this should not be hard.

Roe [31, 32] and Zimmermann [40, 41] both studied profiling semantics for parallel languages. Roe formally defined a profiling semantics for an extended λ-calculus with *lenient* evaluation. In his semantics, the two subexpressions of a special *let* expression *plet* $x = e_1$ *in* e_2 evaluate in parallel such that the evaluation of an occurrence of x in e_2 is delayed until its value is available. To define when this is the case, he augmented the standard denotational semantics with the time that each expression begins and ends evaluation. He did not show any complexity bounds resulting from his definition or relate this model to any other. Zimmerman introduced a profiling semantics for a data-parallel language for the purpose of automatically analyzing PRAM algorithms. The language therefore almost directly modeled the PRAM by

adding a set of PRAM-like primitive operations. Complexity was measured in terms of time and number of processors, as it is measured for the PRAM. It was not shown, however, whether the model exacly modeled the PRAM. In particular since it is not known until execution how many processors are needed, it is not clear whether the scheduling could be done on the fly.

Hudak and X suggest modeling parallelism in functional languages using an extended operational semantics based on partially ordered multisets (pomsets). The semantics can be though of as keeping a trace of the computation as a partial order specifying what had to be computed before what else. Although significantly more complicated, their call-by-value semantics are related to the A-PAL model in the following way. The work in the A-PAL model is within a constant factor of the number of elements in the pomset, and the steps is within a constant factor of the longest chain in the pomset. They did not relate their model to other models of parallelism or describe how it would effect algorithms.

Previous work on formally relating language-based models (languages with cost-augmented semantics) to machine models is sparse. Jones [18] related the time-augmented semantics of simple while-loop language to that of an equivalent machine language in order to study the effect of constant factors in time complexity.

The work-step paradigm has been used for many years for informally describing parallel algorithms [36, 19]. It was first included in a formal model by Blelloch in the VRAM [5]. The NESL language [6], a data-parallel functional language, includes complexity measures based on work and steps and has been used for describing and teaching parallel algorithms. Skillicorn [37] also introduced cost measures specified in terms of work and steps for a data-parallel language based on the Bird-Meertens paradigm. In both cases the languages were not based on the pure λ-calculus but instead included array primitives. Also neither formally showed relationship of their models to machine models. Part of the motivation of the work described in this paper was to formalize the mapping of complexity to machine models and to see how much parallelism is available without adding data-parallel primitives.

Dornic, *et al.* [10] and Reistad and Gifford [30] explore adding time information to a functional language type system. But for type inference to terminate, only special forms of recursion can be treated, such as those of the Bird-Meertens formalism.

There has been much work on comparing machine models within traditional complexity theory. The most closely related is that of Ben-Amram and Galil [2], who show that a pointer machine incurs logarithmic overhead to simulate a RAM. The pointer machine [20, 38] is similar to the SECD machine in that it addresses memory only through pointers, but it lacks direct support for implementing higher-order functions. We borrow from them the parameterization of models over incompressible data types and operations. Paige [25] also compares models similar to those used by Ben-Amram and Galil.

Goodrich and Kosaraju [14] introduced a parallel pointer machine (PPM), but this is quite different from our model since it assumes a fixed number of processors and allows side effecting of pointers. Another parallel version of the SECD machine was introduced by Abramsky and Sykes [1], but their SECD-m machine was non-deterministic and based on the fair merge.

8 Conclusions

This paper has discussed a complexity model based on the λ-calculus and shown various simulation results. A goal of this work is to bring a closer tie between parallel algorithms and functional languages. We beleive that language-based complexity models, such as the ones suggested in this paper, could be a useful way for describing and thinking about parallel algorithms directly, rather than always needing to translate to a machine model.

This paper leaves several open questions. These questions include

- In the introduction we mentioned that a call-by-speculation implementation of normal-order evaluation

might allow for improved step bounds for various problems. In particular it allows for pipelined execution. Does this help, and on what problems?

- Is it possible to sort in $s = o(\log^2 n)$, and $w = O(n \log n)$?
- Can the bounds for simulating the A-PAL on a PRAM be improved? The bounds for the butterfly network are tight.
- Our simulations are memory inefficient. Can good bounds be placed on the use of memory?
- Because of lack of random-access, can the A-PAL model be simulated more efficiently than the PRAM on machines that have less powerful communication (*e.g.*, fixed-topology networks, parallel I/O models, or the LOGP model [9]), and can the complexity model be augmented to capture the notion of locality for these machines?

References

[1] Samson Abramsky and R. Sykes. Secd-m: A virtual machine for applicative programming. In Jean-Pierre Jouannaud, editor, *2nd International Conference on Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 81–98, 1985.

[2] Amir M. Ben-Amram and Zvi Galil. On pointers versus addresses. *Journal of the ACM*, 39(3):617–648, July 1992.

[3] Bror Bjerner and Sören Holmström. A compositional approach to time analysis of first order lazy functional programs. In *4th International Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag, September 1989.

[4] Guy Blelloch. *An Ll User's Manual (Version 1.2: Draft)*, November 1989.

[5] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[6] Guy E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[7] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[8] Richard Cole. Parallel merge sort. In *Proceedings Symposium on Foundations of Computer Science*, pages 511–516, October 1986.

[9] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[10] Vincent Dornic, Pierre Jouvelot, and David K. Gifford. Polymorphic time systems for estimating program complexity. *ACM Letters on Programming Languages and Systems*, 1(1):33–45, March 1992.

[11] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A Report on the Sisal Language Project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, December 1990.

[12] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[13] J. Gil, Yossi Matias, and Uzi Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proceedings Symposium on Foundations of Computer Science*, pages 698–710, October 1991.

[14] Michael T. Goodrich and S. Rao Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. In *30th Annual Symposium on Foundations of Computer Science*, pages 190–195, November 1989.

[15] T. Hagerup. Fast deterministic processor allocation. In *SODA*, 1993.

[16] Paul Hudak and Steve Anderson. Pomset interpretations of parallel functional programs. In *3rd International Conference on Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 234–256. Springer-Verlag, September 1987.

[17] Paul Hudak and Eric Mohr. Graphinators and the Duality of SIMD and MIMD. In *ACM Conference on Lisp and Functional Programming*, pages 224–234, July 1988.

[18] Neil D. Jones. Constant time factors *do* matter (extended abstract). In *25th ACM Symposium on Theory of Computing*, pages 602–611, 1993.

[19] R. M. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science — Volume A: Algorithms and Complexity*. MIT Press, Cambridge, Mass., 1990.

[20] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley Publishing Company, Reading, MA, 1968.

[21] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[22] Yossi Matias and Uzi Vishkin. Converting high probability into nearly-constant time—with applications to parallel hashing. In *Proceedings ACM Symposium on Theory of Computing*, pages 307–316, May 1991.

[23] Kurt Mehlhorn and Uzi Vishkin. Randomized and deterministic simulations of prams by parallel machines with restricted granularity of parallel memory. *Acta Informatica*, 21:339–374, 1984.

[24] Rishiyur S. Nikhil. ID Version 90.0 Reference Manual. Computation Structures Group Memo 284-1, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1990.

[25] R. Paige. Real-time simulation of a set machine on a RAM. In W. Koczkodaj, editor, *International Conference on Computing and Information*, pages 68–73, 1989.

[26] S. L. Peyton Jones. Parallel Implementations of Functional Programming Languages. *The Computer Journal*, 32(2):175–186, 1989.

[27] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[28] Abhiram G. Ranade. How to emulate shared memory. In *Proceedings Symposium on Foundations of Computer Science*, pages 185–194, 1987.

[29] Abhiram G. Ranade. *Fluent Parallel Computation*. PhD thesis, Yale University, Department of Computer Science, New Haven, CT, 1989.

[30] Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. In *ACM Conference on LISP and Functional Programming*, pages 65–78, July 1994.

[31] Paul Roe. Calculating lenient programs' performance. In Simon L Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming, Glasgow 1990*, Workshops in computing. Springer-Verlag, August 1990.

[32] Paul Roe. *Parallel Programming using Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, February 1991.

[33] Mads Rosendahl. Automatic complexity analysis. In *4th International Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag, September 1989.

[34] David Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, University of London, Imperial College, September 1990.

[35] Yossi Shiloach and Uzi Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2(1):88–102, 1981.

[36] Yossi Shiloach and Uzi Vishkin. An $O(n^2 \log n)$ parallel Max-Flow algorithm. *J. Algorithms*, 3:128–146, 1982.

[37] David B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. To appear in the Journal of Parallel and Distributed Computing.

[38] Robert E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. System Sci.*, 18:110–127, 1979.

[39] Philip Wadler. Strictness analysis aids time analysis. In *15th ACM Symposium on Principles of Programming Languages*, January 1988.

[40] Wolf Zimmermann. Automatic worst case complexity analysis of parallel programs. Technical Report TR-90-066, International Computer Science Institute, December 1990.

[41] Wolf Zimmermann. Complexity issues in the design of functional languages with explicit parallelism. In *International Conference on Computer Languages*, pages 34–43, 1992.

A Code for Merging

```
datatype 'a oseq = leaf of 'a | node of int * 'a * 'a oseq * 'a oseq

fun left_trim (node (n,v,l,r),v0) =
    if > v0 (maxval l) then left_trim (r,v0) else mnode (left_trim (l,v0),r)
  | left_trim (leaf v,v0) = leaf v

fun in_range (s,v0,v1) = right_trim (left_trim (s,v0),v1)

fun kth_smallest (leaf v1,leaf v2,k) =
    if > v2 v1 then if = k 0 then v1 else v0
```

```
    else if = k 0 then v0 else v1
  | kth_smallest (leaf v1,node (n2,v2,l2,r2),k) =
    if > v2 v1 then if > k n2
         then kth_smallest(leaf v1,r2,+ k (-n2))
         else kth_smallest(leaf v1,l2,k)
    else if > n2 k
         then kth_smallest(leaf v1,l2,k)
         else kth_smallest(leaf v1,r2,+ k (-n2))
  | kth_smallest (node (n1,v1,l1,r1),leaf v2,k) =
    kth_smallest(leaf v2, node (n1,v1,l1,r1),k) =
  | kth_smallest (node (n1,v1,l1,r1),node (n2,v2,l2,r2),k) =
    if > v2 v1 then if > k (+ n1 n2)
       then kth_smallest (node (n1,v1,l1,r1),l2,k)
       else kth_smallest (r1,node (n2,v2,l2,r2),+ k (-n1))
    else if > k (+ n1 n2)
       then kth_smallest (l1,node (n2,v2,l2,r2),k)
       else kth_smallest (node (n1,v1,l1,r1),r2,+ k (-n1))

fun merge_sort a =
  if > 2 (length a) then a
  else let mid = /2 (length a)
       in parallel_merge (merge_sort (subseq (0,mid,a)),
                          merge_sort (subseq (mid,length a,a)))
```

B Array Extensions to the A-PAL model

In this appendix we extend the A-PAL model with a set of constants and expressions for manipulating arrays.

$$\begin{aligned} c \in \textit{Constants} &::= \ldots \mid \vec{v} \mid \textbf{put} \mid \textbf{elt} \mid \textbf{len} \mid \textbf{index} \\ e \in \textit{Expressions} &::= \ldots \mid \textbf{map}\ e_1\ e_2 \end{aligned}$$

where $\vec{v}$ ranges over arrays $[v_1, \ldots, v_n]$ for any $n \geq 0$. The primitive **put** allows concurrent writes, as

$$\textbf{put}\ [11,33,66,22,55]\ [3,5,3]\ [333,777,999]$$

evaluates to

$$[11,33,333,22,777] \text{ or } [11,33,999,22,777].$$

The values in the third array are put into the first array according to the indices of the second array, with conflicts resolved arbitrarily here. The other primitives extract an element of an array (**elt**), find the length of an array (**len**), and create an index array (**index**). A **map** expression maps a function e_1 element-wise over e_2. These additions are sufficient for most needs.

The following two rules describe **map**. The function to be mapped (e_1) and the argument (e_2) are evaluated in parallel. Then the value of the function, either a closure (MAP) or a constant (MAPC), is applied in parallel to the elements of the value of the argument, which should be an array.

$$\frac{\begin{array}{c} E \vdash e_1 \xrightarrow{\lambda} cl(E', x, e'); s_1, w_1 \qquad E \vdash e_2 \xrightarrow{\lambda} \vec{v'}; s_2, w_2 \\ E'[x \mapsto v'_i] \vdash e' \xrightarrow{\lambda} v_i; s'_i, w'_i \qquad \forall i \in \{1, \ldots, |\vec{v}|\} \end{array}}{E \vdash \textbf{map}\ e_1\ e_2 \xrightarrow{\lambda} \vec{v}; \max(s_1, s_2) + \max_{i=1}^{|\vec{v}|} s'_i + 1, w_1 + w_2 + \sum_{i=1}^{|\vec{v}|} w'_i + 1} \quad \text{(MAP)}$$

$$\frac{E \vdash e_1 \xrightarrow{\lambda} c; s_1, w_1 \qquad E \vdash e_2 \xrightarrow{\lambda} \vec{v'}; s_2, w_2 \qquad \delta(c, v'_i) = v_i \qquad \forall i \in \{1, \ldots, |\vec{v}|\}}{E \vdash \mathbf{map}\ e_1\ e_2 \xrightarrow{\lambda} \vec{v}; \max(s_1, s_2) + 1, w_1 + w_2 + \sum_{i=1}^{|\vec{v}|} \delta_w(c, v'_i) + 1} \quad \text{(MAPC)}$$

$$\begin{array}{rclrcl}
\delta(\mathbf{put}, \vec{v}) & = & \mathbf{put}_{\vec{v}} & \delta_w(\mathbf{put}, \vec{v}) & = & 1 \\
\delta(\mathbf{put}_{\vec{v}}, \vec{i}) & = & \mathbf{put}_{\vec{v},\vec{i}} & \delta_w(\mathbf{put}_{\vec{v}}, \vec{i}) & = & 1 \\
\delta(\mathbf{put}_{\vec{v},\vec{i}}, \vec{v'}) & = & \vec{v}[\vec{v'}/\vec{i}] & \delta_w(\mathbf{put}_{\vec{v},\vec{i}}, \vec{v'}) & = & |\vec{i}| \\
\delta(\mathbf{elt}, \vec{v}) & = & \mathbf{elt}_{\vec{v}} & \delta_w(\mathbf{elt}, \vec{v}) & = & 1 \\
\delta(\mathbf{elt}_{\vec{v}}, i) & = & v_i & \delta_w(\mathbf{elt}_{\vec{v}}, i) & = & 1 \\
\delta(\mathbf{len}, \vec{v}) & = & |\vec{v}| & \delta_w(\mathbf{len}, \vec{v}) & = & 1 \\
\delta(\mathbf{index}, i) & = & [0, \ldots, i-1] & \delta_w(\mathbf{index}, i) & = & i
\end{array}$$

To extend the P-ECD machine to work on this model would require adding the capability of creating multiple states on each step, the ability to do a non-constant amount of work for each state (and balance it), and the ability to synchronize among multiple states at the completion of the map.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890